NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

PSEUDORANDOM NUMBER GENERATORS FOR MOBILE DEVICES: AN EXAMINATION AND ATTEMPT TO IMPROVE RANDOMNESS

by

Ola Larsson

September 2013

Thesis Advisor:	Pantelimon Stanica
Co-Advisor:	Zachary Peterson

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704–0188) Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2013	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE PSEUDORANDOM NUMBER GENERATORS FOR MOBILE DEVICES: AN EXAMINATION AND ATTEMPT TO IMPROVE RANDOMNESS			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ola Larsson			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis examines the quality of pseudorandom number generation for cryptographic purposes in general and the generation of such numbers in a mobile device (Android phone), in particular, since we expected to find non-random properties in these.

Initially, the need for random numbers for encryption purposes is discussed from a perspective of Information Warfare. Thereafter, ways of testing a bit string for random properties as well as some pseudorandom number generating algorithms are presented. This also includes the shrinking and the self-shrinking generator normally used to improve the random properties of the output m-sequence of linear feedback shift registers. A couple of possible attacks on pseudorandom number generators are also briefly presented.

Finally, we generate and analyze some pseudorandom bit strings in three different ways using the NIST test suite, both before and after the self-shrinking generator has been applied to them. The strings generated by the Android phone passed the NIST test suite, and it is difficult to claim any improvement in random properties by applying the self-shrinking generator. On a bit string with poor random properties, however, the self-shrinking generator improves randomness from the perspective of linear dependency and complexity, but not from the perspective of bit frequency.

14. SUBJECT TERMS Pseudorandom number generator, PRNG, Random number, Random bit, Self-shrinking generator, SSG, Encryption, Mobile device, Android			**15. NUMBER OF PAGES** 77
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01–280–5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

PSEUDORANDOM NUMBER GENERATORS FOR MOBILE DEVICES: AN EXAMINATION AND ATTEMPT TO IMPROVE RANDOMNESS

Ola Larsson
Major, Swedish Army
M.S., Chalmers University of Technology, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION WARFARE SYSTEMS ENGINEERING

AND

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
September 2013

Author:	Ola Larsson
Approved by:	Pantelimon Stanica Thesis Advisor
	Zachary Peterson Co-Advisor
	Raymond Buettner Second Reader
	Dan Boger Chair, Department of Information Sciences
	Carlos Borges Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis examines the quality of pseudorandom number generation for cryptographic purposes in general and the generation of such numbers in a mobile device (Android phone), in particular, since we expected to find non-random properties in these.

Initially, the need for random numbers for encryption purposes is discussed from a perspective of Information Warfare. Thereafter, ways of testing a bit string for random properties as well as some pseudorandom number generating algorithms are presented. This also includes the shrinking and the self-shrinking generator normally used to improve the random properties of the output *m*-sequence of linear feedback shift registers. A couple of possible attacks on pseudorandom number generators are also briefly presented.

Finally, we generate and analyze some pseudorandom bit strings in three different ways using the NIST test suite, both before and after the self-shrinking generator has been applied to them. The strings generated by the Android phone passed the NIST test suite, and it is difficult to claim any improvement in random properties by applying the self-shrinking generator. On a bit string with poor random properties, however, the self-shrinking generator improves randomness from the perspective of linear dependency and complexity, but not from the perspective of bit frequency.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ASCII	American Standard Code for Information Interchange
BBS	Blum-Blum-Shub (pseudorandom number generator)
DoD	Department of Defense
EA	Electronic Attack
EP	Electronic Protection
ES	Electronic Warfare Support
EW	Electronic Warfare
GF	Galois Field
IW	Information Warfare
LSFR	Linear Feedback Shift Register
MAC	Message Authentication Code
MISO	Military Information Support Operations
NIST	National Institute for Standards and Technology
OPSEC	Operational Security
PRBG	Pseudorandom Bit Generator
PRNG	Pseudorandom Number Generator
PSYOP	Psychological Operations
XOR	Exclusive or (mathematical operation)

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First of all, I want to thank my advisors, Pantelimon Stanica and Zachary Peterson, who helped me turn my early thesis idea into a suitable project and who also helped guide me along the way, giving me useful feedback and comments, even at the very end with tight time constraints. Your patience, time and effort have been highly appreciated, thank you! I also want to thank Mark Gondree for invaluable help at a critical moment at a late stage. The support from my second reader, Raymond Buettner, and my two Program Officers, William Robinette and Thor Martinsen, have also been a great help. But really, all the professors and lecturers that I've had the privilege to study with during my time at the Naval Postgraduate School have become a part of me, through my studies and this thesis, and now in earning my degrees.

Finally, my studies would have been so much tougher without having you by my side, Frida. Thank you for keeping me afloat when the waters have been rough!

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

When the average man thinks about war and warfare, the first thing that comes into his mind might be images of traditional wars like World War II. Wars in which battles were fought on a distinct battleground, where man fought against man, tank against tack, airplane against airplane and ship against ship. Battles like these are easy to understand. The way to defeat your opponent is to destroy him through physical means, and it is easy to see who walks away from a duel a winner, and who the loser. What many might tend to forget is that there was a long series of events and processes leading up to every battle. Battles were never fought by coincidence, at least one side had knowledge of what was about to happen and believed it could gain from it; we typically call this knowledge intelligence.

Intelligence has always played an important role in warfare. With knowledge of your own forces, and good and reliable intelligence regarding your opponent, you can choose when and where to engage with him in a battle. You also know what to expect from your opponent, what resources he has, the morale of his troops, his ideas, tactics and operational skills, his strengths and his weaknesses; in short, you know everything that affects his possibility to fight you. Sun Tzu said in one of his most famous quotes: "So it is said that if you know others and know yourself, you will not be imperiled in a hundred battles; if you do not know others but know yourself, you win one and lose one; if you do not know others and do not know yourself, you will be imperiled in every single battle."[1] Of course, intelligence is not always easily collected. Perhaps even more importantly, as Clausewitz stated, it is not always easily interpreted and used.[2] A military power that can use intelligence to its advantage, and also control the opponent's access to intelligence, has a big advantage and can successfully conduct large-scale war-changing operations like the invasion in Normandy 1944. Through superior intelligence capabilities, the

[1] Sun Tzu, *The Art of War,* Trans. Thomas Cleary (Boston, MA: Shambhala Publications, 1991), 24.

[2] Carl von Clausewitz, *Om Kriget*, Trans. Hjalmar Mårtensson, Klaus-Richard Böhme and Alf W Johansson (Stockholm, Sweden: Bonnier Fakta Bokförlag AB, 2002) 77–78.

advantage in information available and possibilities to plant false intelligence into the German intelligence service, the Allied forces could deceive Hitler, thereby creating favorable preconditions for an amphibious landing and taking a big step towards ending a war that had been tormenting Europe for five years.[3]

Kinetic destruction, as in World War II, still plays an important role in today's warfare. But with the introduction of modern technology the possibilities for gaining intelligence have changed dramatically. This technology makes it possible for us to collect and get access to important intelligence to a greater extent than ever before. This also means, however, that our opponent has the same possibilities. We therefore have to protect our own sensitive data and information carefully. This might be more difficult than one would first think, since almost all information regarding our forces—their capabilities, equipment, locations, actions and interactions—could be of interest to an adversary. Certainly, information has become one of the cornerstones of modern warfare.

A. INFORMATION WARFARE

To better explain the importance of information in today's military, the term Information Warfare (IW) has been introduced. One definition of Information Warfare by the United States Department of Defense (DoD) is as follows:

"Information warfare includes actions taken to preserve the integrity of one's own information systems from exploitation, corruption, or disruption, while at the same time exploiting, corrupting, or destroying an adversary's information systems and the process achieving an information advantage in the application of force."[4]

Furthermore, information warfare can be described as a structure with five pillars with a common foundation of intelligence. The five pillars consist of Psychological Operations (PSYOP), Deception, Electronic Warfare (EW), Destruction and Operational

[3] William B. Breuer, *Hoodwinking Hitler, the Normandy Deception* (Westport, CT: Praeger Publishers, 1993).

[4] Edward Waltz, *Information Warfare, Principles and Operations* (Norwood, MA: Artech House, Inc, 1998), 20.

Security (OPSEC).[5] These are five free standing columns, but they can often be used in interaction with each other.

1. Psychological Operations

Psychological operations are the operations to try to affect an opponent by influencing his emotions, reasoning and behavior. This is done through our own troops actively spreading information favorable to us but misleading to an adversary, reinforcing his misinterpretations and misleading him in his estimations about what damages he has caused us. PSYOP is often conducted through traditional open source media like radio, TV and printed news, but can also be more directed through leaflets and media campaigns in conflict areas. PSYOP is often directed at the adversary's civilian population to try to disrupt the people's support of their leaders or to encourage them to revolt.[6]

2. Deception

Deception is the military act of actively misleading an enemy, putting him in a situation where he believes that he has a correct image of the situation while the correct image really is very different. This has been done by armies throughout history; some have managed better and some worse. To be successful one has to perform a deception that gives an adversary information inputs from multiple sources, all confirming each other. But one also has to make sure that information revealing the true image is not accessible. The purpose of deception is to create a situation where the opponent is engaged in actions that will not interfere with ours and restrain him from taking actions on our movements and attacks.

[5] David L. Adamy, *EW102, A Second Course in Electronic Warfare* (Boston, MA: Artech House Publishers, 2004), 5–6.

[6] The term "Psychological Operations" have been changed to "MISO – Military Information Support Operations." The older term is however still commonly used. United States Joint Chiefs of Staff, Joint Publication 3–13.2. *Military Information Support Operations*. (Washington, DC, 2011).

3. Electronic Warfare

Electronic Warfare is the use of the electromagnetic spectra for military success. It is divided into three subgroups: Electronic Attack (EA), Electronic Warfare Support (ES) and Electronic Protection (EP). EW includes techniques and activities such as analysis of the electromagnetic spectra (i.e., what frequencies are being used and for what), analysis of how an opponent's wireless communication network is constructed, attacking weapons and sensor systems that actively or passively are used in the electromagnetic spectra and also protecting ourselves against similar actions and attacks.

4. Destruction

Destruction in this context refers to the destruction of information warfare capabilities. Destruction of information and intelligence, electromagnetic structures like radar systems, communication nodes and other means of communication. Destruction can be accomplished not only through kinetic energy at relatively close distance but also through non-kinetic energy at great distances, e.g., through cyber-attacks on computer networks to erase and destroy crucial data.

5. Operational Security

Operational security is intended to protect information about our resources, aims, intentions, etc., from falling into the hands of our adversary. Just as we try to get as much information about our opponent, he tries just as hard to get to know about us. To conduct successful operations with a minimum of losses we have to make sure that our opponent is denied this information. Maintaining a high operational security is of the utmost importance to achieve our goals.

B. PROTECTION OF INFORMATION

From what has previously been discussed it is clear that in information warfare, protection of information is of great importance. Looking at the five pillars of information warfare, at least two pillars, Deception and OPSEC, have a direct need to have the means of protecting information. Protecting information can be accomplished in many ways, one of which is making it non-accessible by locking it into a vault. But we not only want to protect information from falling in the hands of an enemy, we also want to share the same information within our forces and to friendly forces. Thus, we need ways to securely communicate information. The means of doing so is called cryptology.

Cryptology (or cryptography, from Greek: cryptos = “hidden/secret” and –logia = “study” or graphein = “writing”) has been used for military purposes for a long time. Early encryption methods are the transposition cipher where letters are rearranged or the substitution cipher where every letter is represented by another letter making the message unreadable if you do not know the method used (the key). Some examples of these classical cryptology schemes are the Caesar and the Vigenère ciphers.[7] Today we use more sophisticated methods of encryption but the basic idea is still the same; we want to transmit a plain text message securely by applying an encryption algorithm. When it comes to the message, we want to be able to transmit any plain text message without limitations; i.e., we do not want to have to adjust our plain text message to fit the encryption algorithm in any way. For the encryption algorithm itself we nowadays assume “Kerckhoff’s principle,” assuming that the algorithm itself is commonly known and cannot be used as the sole means of protecting the message.[8] Since we now have two entities, none of which we can modify to gain protection we use a third entity to do so, namely the encryption key.

[7] For an excellent overview of the history and development of encryption, see Simon Singh, *The Code Book* (New York, NY: Random House 1999).

[8] Douglas R. Stinson, *Cryptography Theory and Practice*, 3rd ed. (Boca Raton, FL: Chapman & Hall/CRC, 2006), 26.

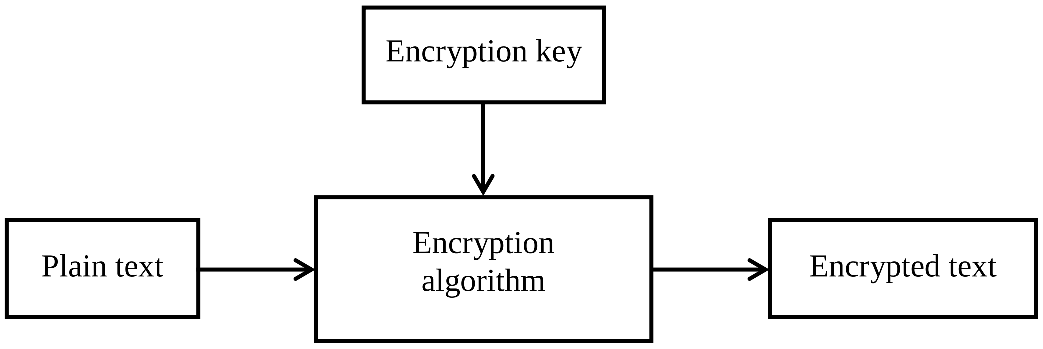


Figure 1. Plain text and encryption key used as an input to an encryption algorithm resulting in an encrypted text.

The encryption key is what the encryption algorithm uses as an initial input to start encrypting the plain text. One can see the key being the initial settings of a number of variables in a very complicated machine that for each iteration changes according to their previous values. All changes to the settings are deterministic and depend on their current value, and the plain text is entered so it is only the initial value that affects the changes. Therefore, it is of utmost importance that the key (or the initial setting) is chosen in such a way that no one can guess or gain access to it. If a human would be given the responsibility to choose an encryption key, he would most likely choose a key that would be easy to remember (e.g., for decryption purposes), just like most people do when they choose a password for their online services. Just as passwords can be broken using regular dictionary lookups, password dictionary lookups,[9] or trying simple substitution methods (like using common words but changing the letter “O” to a zero) encryption keys chosen by humans could be broken just as easily. To prevent this we choose numerical encryption keys randomly. Since a human is not good at picking random numbers we have to rely on machines to create random numbers or strings of random numbers.

[9] Online you can find dictionaries with passwords that have been broken or in other ways obtained through attacks on numerous databases. Any password that has ever been broken, therefore, has just as bad reliability as any dictionary word.

C. RANDOM NUMBERS

To most people, creating random numbers might seem to be one of the easiest tasks there is. You do not have to think, just pick a number from among several others. However, on the contrary, it is extremely difficult to do so. By randomness, we normally mean non-predictable; i.e., among a given number of alternative outcomes (where only one can occur) all outcomes should have the same likeliness of happening. A person participating in a raffle, for example, would expect to have the same chance of winning the grand prize as anyone else. In a raffle, randomness is often achieved by tickets being mixed in a container with the winning ticket being drawn by an official. But what if the tickets are not properly mixed and instead are just put in a jar as they are being sold? What if the lottery tickets have different sizes, weights, paper quality or colors? This does not automatically mean that a winning ticket cannot be drawn at random, but the chances of the raffle official being influenced by such factors and thereby biased increases. If the raffle official knows ahead of time that the tickets differ and also knows which type of tickets belong to which participants he has a greater possibility to affect the outcome of the raffle. This would not elect the winner of the raffle randomly, and the lottery would not be considered fair.

In the situation described in the previous paragraph, the raffle official drawing the winning ticket is clearly a great risk to biasing the outcome. Therefore, mechanical raffle and lottery machines are widely used in state arranged lotteries like the Mega Millions, Powerball and Lotto. Mechanical lottery machines normally just draw numbers identifying the winning ticket/tickets and use either gravity or air flow to pick a ball indicating a number in the appropriate range. Depending on the type of lottery, the drawn ball is either put aside or put back to make it possible to be drawn again. This is an illustrative way of picking numbers and given that each number is represented on one of the balls, all balls are of the same weight and size and are properly mixed, this is a fair way to pick random numbers.

Even if mechanical machines are good from a perspective of picking random numbers in a fair way, they are not very practical when it comes to computer applications. They are just too large and too slow, and of course do not produce a usable digital output. In the early days of computer programming users started to search for efficient ways to generate random numbers using computers. John von Neumann created the “middle-square” method, one of the first methods to be used. In this method a random number is generated by taking the previous random number, squaring it and extracting the middle digits of the result. Von Neumann used 10 digits, while others suggested both more and fewer digits.[10] The problem with the “middle-number” method is that it is not a very good random number generator; the numbers achieved just appear to be random. When analyzed mathematically, it is clear that they lack important properties of random numbers. Furthermore the randomness of the output greatly depends on which input is being used. There are many examples of inputs that quite soon will “loop” and get back to an already used “random” number; i.e., they have a very short period. Others do, however, result in random numbers that pass appropriate statistical tests.

Random numbers are today used in a number of different areas such as simulation, sampling, numerical analysis, computer programming and recreation. This thesis will focus on the generation and use of random numbers for encryption purposes. Since we normally use computers for encryption it is not necessary to create random decimal numbers; binary bit strings will do just fine. All that is needed is an electronic “coin tosser” creating random “heads and tails” interpreted as 1’s and 0’s, or in other words; something that “assigns a numerical value to the outcome of the random experiment.”[11] To be able to implement such a feature in a computer or a mobile device we have to make it computable; i.e., we need an algorithm to do this for us. Such an algorithm will give us a pseudorandom number generator (PRNG). A PRNG is said to be pseudorandom since the output is not actually random; it is the result of a mathematical computation performed by a deterministic machine operating under given circumstances. A truly random number generator would not use any computations at all, thereby being

[10] Donald E. Knuth, *The Art of Computer Programming, Volume 2/Seminumerical Programming*, 2nd ed. (Reading, MA: Addison-Wesley Pub. Co., 1981), 3–4.

[11] Alberto Leon-Garcia, *Probability and Random Processes for Electrical Engineering*, 2nd ed. (Reading MA: Addison-Wesley Pub. Co., 1994), 84.

totally unpredictable. By choosing good algorithms with proper complexity, a PRNG can be created whose output has the properties of being random. So even if by using a PRNG we compute a string of random bits, thereby making the output predictable, this string has (or should have) the same properties of a truly random string. The output depends on the input and the algorithms used. The challenge is to create a PRNG that creates bit strings with properties of random numbers and also does not reveal any information on the data used as the input creating these strings.

THIS PAGE INTENTIONALLY LEFT BLANK

II. TESTING FOR RANDOMNESS

Testing to see whether a bit string is random or not can be challenging since there are many ways in which non-randomness can appear. A string having approximately the same number of ones and zeroes is perhaps an obvious test for a random bit string, but one also has to take into consideration other aspects such as repeating patterns, length of runs (repeating bits), linear dependency, etc.

A. THE NIST TEST SUITE

The U.S. National Institute for Standards and Technology (NIST) has developed a suite of random number generation tests. This test suite is available for download at the NIST homepage[12] together with a thorough description on how the tests work, how they should be applied and how the results can be interpreted. The suite consists of 15 different tests examining different aspects of randomness of a binary sequence. The purpose of these tests is to support the user in deciding whether a sequence is random or not. NIST does not claim that a sequence passing the tests in the suite really is random; it is always up to the user to interpret the test results and make that decision himself or herself based on the results.

To run the tests the user needs a chosen number of generated bit strings of equal lengths to be tested for randomness. Each such string is to be treated as being one sequence in a longer file of sequences. Therefore, the test suite needs both the sequence length and the number of sequences to be tested as an input. No recommendations regarding number of sequences is given, but for the test results presented later 100 bit strings were used. The 15 tests in the NIST test suite are presented here and briefly explained[13]:

[12] http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html.

[13] Andrew Rukhin et al., *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application* (National Institute of Standards and Technology (NIST) Special Publication 800–22 Rev. 1a), (Gaithersburg, MD: U.S. Department of Commerce, 2010).

1. The Frequency (Monobit) Test

This test checks the occurrence of “1” and “0” throughout the full sequence. In a truly random sequence we would expect about half of the bits to be 1 and the other half to be 0. This test checks if the test sequence diverges too far from this. The test is the most basic test of the fifteen. If a bit string does not pass this test, it is barely worth running the other tests. It can be seen as serving as a basis for all other tests in the test suite.

2. The Frequency Test within a Block

This tests the occurrence of 1 and 0 within blocks of the same size. In a truly random sequence we would expect about half of the bits in each block to be “1” and the other half to be “0.” This test is the same test as the previous one; it is just limited to blocks of a given size M. The block size M can be chosen by the user.

3. The Runs Test

This tests the length of runs in the full sequence. A run is defined as an unchanged sequence of bits bounded by differing bits. In other words, it could be described as the rate at which the bits alternate within the sequence. The statistical possibility of a bit being the same as the previous one is ½ in a truly random sequence. The chance at any given time of having a run of length n is $(½)^n$.

4. The Test for the Longest-Run-of-Ones in a Block

This tests the length of runs of “1” in blocks of the same size M. The block size tested depends on the length of the total sequence and can be chosen to be one of three different preset sizes. Even if the test only checks for runs containing “1” an indicated lack of randomness of the number of runs of “1” indicates an equivalent lack of randomness for the number of runs of “0.”

5. The Binary Matrix Rank Test

This test divides the sequence into matrices of a given size and checks these matrices individually for linear dependencies. To pass the test the created matrices must have a high level of linear independency.

6. The Discrete Fourier Transform (Spectral) Test

This test checks the peaks in the Discrete Fast Fourier Transform of the full binary sequence. Doing so makes it possible to identify frequency patterns that would indicate non randomness in the sequence.

7. The Non-overlapping Template Matching Test

The test divides the sequence into blocks and bitwise checks each block for the number of occurrences of pre-specified target strings. Once a target string is found, the test continues searching for the next string after the last bit in the string found. Any target string should appear equally often in all blocks.

8. The Overlapping Template Matching Test

This test is similar to the previous test, but once a target string is found, the test starts to search for the next target string on the following bit (i.e., it does not skip to the bit following the last one in the target string). Any target string should appear equally often in all blocks.

9. Maurer’s “Universal Statistical” Test

This test checks the number of bits between matching patterns. This gives an indication of how much the sequence can be compressed. If it is possible to highly compress a sequence, then it might be non-random.

10. The Linear Complexity Test

This test identifies linear dependence in a sequence; do parts of the sequence have a linear dependency on other parts? The test is based on the Berlekamp-Massey

Algorithm for Linear Feedback Shift Registers (LFSRs) described in Chapter IV, Section B of this thesis. A highly complex LSFR (long LSFR) indicates a higher level of randomness.

11. The Serial Test

The test takes a number of m-bit strings and checks the occurrence of these in the tested sequence. In a truly random string all different m-bit strings should occur about equally as often. This test is similar to the Frequency Test (1) but for strings instead of single bits.

12. The Approximate Entropy Test

This test works as the serial test (11) but instead of looking at the whole sequence it looks at two adjacent blocks of the sequence and compares the occurrence of strings in these blocks. The strings are expected to occur about the same number of times in both the blocks.

13. The Cumulative Sums (Cumsums) Test

The test counts “1” as +1 and “0” as -1. Then it checks the cumulative sum of strings of increasing size as it steps through the tested sequence. This test is performed both forward and backwards in the sequence. For the tested sequence to be considered random, its cumulative sums should not deviate too far from zero.

14. The Random Excursions Test

The test checks the value of the cumulative sum in each cycle (a cycle being the period between two cumulative sums being equal to zero). In how many cycles does it hit exactly one of eight given values? Either each sum should be hit very frequently or all sums should be hit just as frequently. Any deviations from this indicate non-randomness.

15. The Random Excursions Variant Test

This test is similar to the previous one when it checks the value of the cumulative sum in each cycle (a cycle being the period between two cumulative sums being equal to zero). Now this test instead checks how often the cumulative sum hits one of the 18 predefined defined values.

B. PRESENTATION OF TEST RESULTS USING THE NIST TEST SUITE

Each test is a statistical hypothesis test in which the null hypothesis (H_0) is that the tested bit string really is random. A test statistic is calculated from the data resulting from the test, and this test statistic is then used to calculate a P-value summarizing the test.[14] The P-value indicates the probability for a truly random number generator generating a sequence less random than the tested one for that specific test. A low P-value (below 0.01 in the test results later presented) means that the null hypothesis should be rejected; i.e., the bit string is not random.

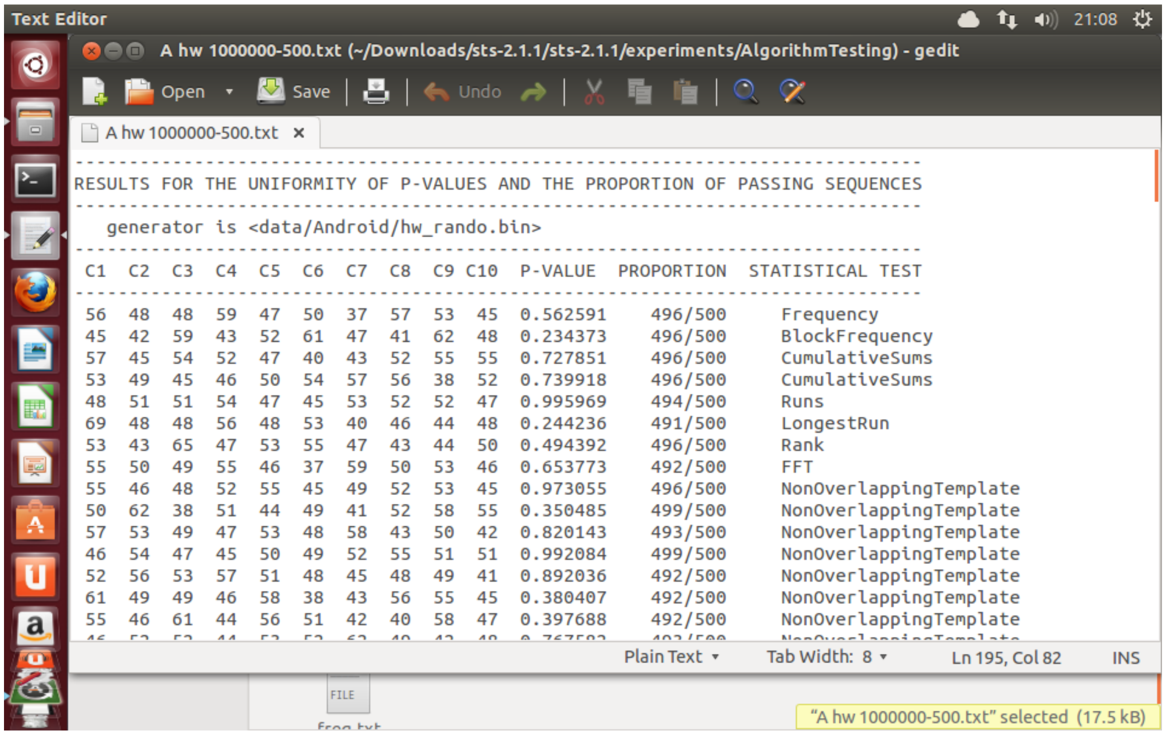
An example of the presented results can be seen in Figure 2. The rightmost column “Statistical test” states to which test the result refers. The column “Proportion” shows how many of the tested strings pass the test. Note that all strings do not have to pass a test for the whole sequence to pass. (True randomness must allow for something to sometimes appear nonrandom.) With the significance level set to 0.01, 1% of the strings can be expected to fail the test. In the result presented in Figure 2 we can see that 500 strings have been tested and depending on which test we look at, somewhere between 491 and 499 strings have passed the tests.

Another way to interpret the results is to look at the distribution of the P-values. Columns “C1” through “C10” indicate ten subintervals of the interval 0 to 1. A P-value is calculated for each tested string (500 in the following example). All P-values fall within one of the ten subintervals and is presented accordingly. For the full sequence of strings to be random, the P-values should be evenly distributed; i.e., there should be about as many in each subinterval. A P-value of this distribution is also calculated and presented

[14] For a more thorough explanation, see Andrew Rukhin et al., *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application* (National Institute of Standards and Technology (NIST) Special Publication 800–22 Rev. 1a), (Gaithersburg, MD: U.S. Department of Commerce, 2010).

in the column “P-value.” Any value exceeding 0.0001 indicates the distribution can be considered evenly distributed.

If a sequence of strings should not pass a test this would be indicated by an asterisk (*) in the “Proportion” and/or “P-value” column. As mentioned earlier, it is ultimately up to the user to decide whether a sequence should be considered to be random or not. The NIST test suite is just an aid to make that decision.


Text Editor
21:08
A hw 1000000-500.txt (~/Downloads/sts-2.1.1/sts-2.1.1/experiments/AlgorithmTesting) - gedit
Open
Save
Undo
A hw 1000000-500.txt
Plain Text
Tab Width: 8
Ln 195, Col 82
INS
FILE
“A hw 1000000-500.txt” selected (17.5 kB)

```
------------------------------------------------------------------------------
RESULTS FOR THE UNIFORMITY OF P-VALUES AND THE PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------------------
   generator is <data/Android/hw_rando.bin>
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 56  48  48  59  47  50  37  57  53  45  0.562591    496/500     Frequency
 45  42  59  43  52  61  47  41  62  48  0.234373    496/500     BlockFrequency
 57  45  54  52  47  40  43  52  55  55  0.727851    496/500     CumulativeSums
 53  49  45  46  50  54  57  56  38  52  0.739918    496/500     CumulativeSums
 48  51  51  54  47  45  53  52  52  47  0.995969    494/500     Runs
 69  48  48  56  48  53  40  46  44  48  0.244236    491/500     LongestRun
 53  43  65  47  53  55  47  43  44  50  0.494392    496/500     Rank
 55  50  49  55  46  37  59  50  53  46  0.653773    492/500     FFT
 55  46  48  52  55  45  49  52  53  45  0.973055    496/500     NonOverlappingTemplate
 50  62  38  51  44  49  41  52  58  55  0.350485    499/500     NonOverlappingTemplate
 57  53  49  47  53  48  58  43  50  42  0.820143    493/500     NonOverlappingTemplate
 46  54  47  45  50  49  52  55  51  51  0.992084    499/500     NonOverlappingTemplate
 52  56  53  57  51  48  45  48  49  41  0.892036    492/500     NonOverlappingTemplate
 61  49  49  46  58  38  43  56  55  45  0.380407    492/500     NonOverlappingTemplate
 55  46  61  44  56  51  42  40  58  47  0.397688    492/500     NonOverlappingTemplate
```

Figure 2. Example of a presentation of NIST test suite results.

C. OTHER RANDOM NUMBER TESTS

For the purpose of testing randomness there are a number of software packages available; NIST, DieHard, DieHarder, TestU01 and ENT are some commonly used. Most of these consist of a test suite in which each test measures a specific aspect of randomness in a bit string. Some tests are the same for the different software packages while others are unique for every test suite. It is up to each creator of the test suite to decide which tests should be included or not as there is no set standard.

III. GENERATING RANDOM NUMBERS

There are many different ways to generate random numbers, from mechanical to computational. Different methods also differ in result; some generate numbers with good random properties while others generate numbers that are not always so random. Depending on how they are to be used, these lesser random numbers can still be acceptable depending on what we want to use them for. Some methods have the sole purpose of generating random numbers while others are not primarily meant to be used for this purpose but can still be quite usable to generate random numbers that do not necessarily have to be cryptographic secure (e.g., for simulation purposes).

A. CRYPTOGRAPHIC HASH FUNCTIONS

A common function resulting in a string with properties of being random is the cryptographic hash function. It is often used for confirming that two files/texts/passwords are identical without comparing them character by character (e.g., for storing digital passwords for online services or for detecting if a text has been modified). A cryptographic hash function takes a clear text as an input to a standardized computation and outputs a fixed length, so-called digest, that appears to be random and in no way reveals the text used as the input.[15]

The strengths of these cryptographic hash functions are that any input results in a fixed length output string of bits. Also, a small change in the input results in a large change in the output (the so called "Avalanche Effect") it is therefore easy to generate a new string with other random properties. Since the result of a cryptographic hash function should have the properties of a random number/string it may be used as a key for an encryption algorithm. The problem is however that we may want longer key strings of pseudorandom data than a hash function alone can provide. To receive a long string of random numbers we instead use pseudorandom number generators.

[15] William Stallings and Lawrie Brown, *Computer Security, Principles and Practice* (Upper Saddle River, NJ: Pearson Educational, 2008), 54–56.

B. MODERN PSEUDORANDOM NUMBERS GENERATORS

Encryption algorithms are, just like computers, deterministic in that sense that given the same input and using the same algorithms we will always receive the same result. The output of an encryption depends on the inputs: the clear text and the encryption key. Therefore, there are three variables: the plain text, the encryption key and the encryption algorithm. The encryption of the plain text lies in its computation through the encryption algorithm using a specific key. According to Kerckhoff's principle[16] the encryption algorithm is assumed to be commonly known, this is an assumption made for all encryption algorithms. This then results in the only two unknown variables being the plain text and the encryption key. The protection of the encrypted plain text, therefore, solely lies in the encryption key, and it is of utmost importance that the encryption key is chosen in a proper way to ensure the secrecy of the encrypted plain text.

The best way of creating a usable encryption key is to use a random number generator. Since computers use binary numbers a simple coin toss with a fair coin (where heads result in a "1" and tails result in a "0") would be a cryptographically good way to create an encryption key. However, in reality this is, of course, not practically usable since we want a long string of random bits, and we also want it generated quickly. Instead we take a random number and use this as an input to a pseudorandom number/bit generator which creates a longer string of bits that we can use for the encryption key.

The pseudorandom number/bit generator is, just as the encryption algorithm, also a deterministic algorithm; given a certain input the same output is always achieved. However, with a good enough pseudorandom number/bit generator a relatively small input of random bits will result in a much longer output that might not be completely random but has most (or, hopefully, all) of the characteristics a truly random bit string has. If an output can be achieved where the likeliness of telling the achieved bit string from a truly random string is less than half the achieved bit string is as good as random and can be used for cryptographic purposes.

[16] Stinson, *Cryptography Theory and Practice*, 26.

There are a number of pseudorandom number/bit generators available. In general, they can be divided into two groups, cryptographic secure and cryptographic insecure. The insecure generators can still be very useful in other areas since they are often easier to implement and faster than the secure generators

1. The Linear Feedback Shift Registers

Although they are linear and thereby predictable and not considered cryptographically secure, LFSRs are still commonly used as pseudorandom number generators. The main reason for this is that they are easily implemented in hardware and therefore very fast to use. With an LSFR, an irreducible (or, better yet, primitive) polynomial[17] is used to create a register. For each clock cycle the bits in the register are moved over one step with the bit in the last position looping back to the first position. At given positions in the register bit information is extracted or inserted to affect the result in the register. There are two main ways to construct LSFR, the Galois and the Fibonacci configurations.

a. Galois LSFRs

In this configuration, the register's positions are numbered from the right to the left. It is started with a seed of all "0's" except for one "1" (in the leftmost position). At each clock cycle the bits are moved over to the right, and the rightmost bit is looped back to the leftmost position. When this rightmost bit is looped back it is XORed with bits from other positions (positions given by the irreducible polynomial) affecting the resulting bits in other positions of the register. Every clock cycle will create a new register content until all possible combinations have been achieved. Using an irreducible polynomial ensures that no combinations are missed (except all "0's" that will generate nothing else but "0's").

[17] See Appendix B.

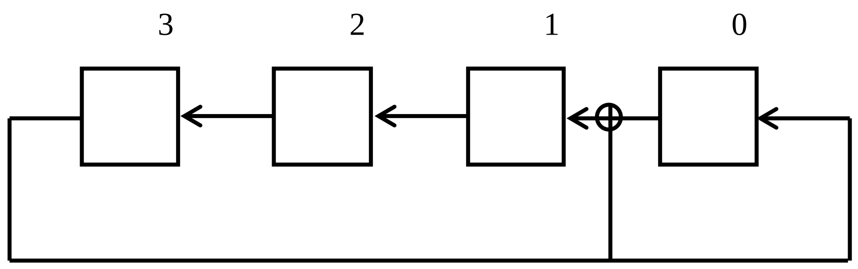


Figure 3. Galois Linear Feedback Shift Register (LSFR) for the generating function $f\left(x\right)=x^4+x+1$.

b. Fibonacci LSFRs

In this configuration, the register's positions are numbered from the left to the right. It is started with a seed of all "0's" except for one "1" (in the leftmost position). At each clock cycle the bits are moved over to the right, and the rightmost bit is looped back to the leftmost position. As the bits are shifted over to the right they are at given positions (positions given by the irreducible polynomial) extracted and XORed with the rightmost bit as it is being looped back to the leftmost position. Every clock cycle will create a new register content until all possible combinations have been achieved. Using an irreducible polynomial ensures that no combinations are missed (except all "0's" that will generate nothing else but "0's").

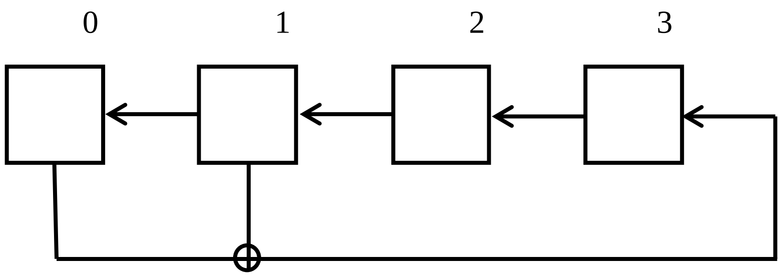


Figure 4. Fibonacci Linear Feedback Shift Register (LSFR) for the generating function $f\left(x\right)=x^4+x+1$.

The result of the registers being run until their content is repeated can be presented as a list of all possible combinations of the bit positions in the registers where the columns represent the different numbered positions in the register, and the rows represent the clock cycles (see Table 1). All these columns are alike, although they are shifted. Any column can now be used as an *m*-sequence with properties of a pseudorandom bit string. The order of “1’s” and “0’s” in the *m*-sequence depends on which irreducible polynomial was chosen for the initial construction of the register. Since all possible combinations of the bits in the registers are generated there will be just as many “1’s” as “0’s” in the *m*-sequence with an exception of one zero. Since all “0’s” will never appear in the registers there will always be one more “1” than “0’s” in the *m*-sequence.

Galois LSFR				
Iteration #	**Position**			
	3	**2**	**1**	**0**
1	0	0	0	1
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0
5	0	0	1	1
6	0	1	1	0
7	1	1	0	0
8	1	0	1	1
9	0	1	0	1
10	1	0	1	0
11	0	1	1	1
12	1	1	1	0
13	1	1	1	1
14	1	1	0	1
15	1	0	0	1

Fibonacci LSFR				
Iteration #	**Position**			
	0	**1**	**2**	**3**
1	1	0	0	0
2	0	0	0	1
3	0	0	1	0
4	0	1	0	0
5	1	0	0	1
6	0	0	1	1
7	0	1	1	0
8	1	1	0	1
9	1	0	1	0
10	0	1	0	1
11	1	0	1	1
12	0	1	1	1
13	1	1	1	1
14	1	1	1	0
15	1	1	0	0

Table 1. List of combinations for Galois and Fibonacci LSFRs using the generating function $f(x) = x^4 + x + 1$.Note the reverse order of the positions but how the *m*-sequences (shaded) are equal. Also note how in the Fibonacci LSFR the sequences at the different positions are the same, just shifted.

2. The Linear Congruential Generator

The linear congruential generator is a fast, widely used generator that is however insecure. It was introduced in 1949 by D. H. Lehmer and was for a long time the most popular random number generator.[18] The generator uses two constants, *a* and *b*, within a defined interval, a modulus, *M*, and a randomly chosen seed, *x*. The first random number is achieved by multiplying the constant *a* by the seed *x*, adding the constant *b* and taking the result modulo *M*.

[18] Knuth, T*he Art of Computer Programming*, 9.

M, a real number

a and *b*, constants such that $1 \le a, b \le M - 1$

x, seed such that $0 \le x \le M - 1$

$x_1 = (a \cdot x + b) \bmod M$

From this computation the least significant bit is used as the first random bit (which is the same thing as taking the result modulo 2). For the following bit the number of the computation in the previous step is used instead of the seed. This process is repeated until required number of bits has been achieved.

$$x_i = (a \cdot x_{i+1} + b) \bmod M$$

Further limitations for the generator are that the seed should be a bit string of length no longer than length *k* and no pseudo random bit strings longer than length *l* should be generated where *k* and *l* are defined as follows:

$$k = 1 + \log_2 \lfloor M \rfloor$$

$$k + 1 \le 1 \le M - 1$$

The resulting generator is then called a (*k*,*l*)-linear congruential generator

The linear congruential generator is not considered secure enough and should be avoided for cryptographic purposes since it can be predictable.[19] Since it is easy to compute and implement and also very fast, though, it is widely usable in other areas where security is of lesser importance (e.g., games, simulations, etc.). The linear congruential generator repeats itself (i.e., it is periodic) after a certain number of iterations, and some seeds result in a shorter repetition period than others.

[19] Wade Trappe and Lawrence C. Washington, *Introduction to Cryptography with Coding Theory*, 2nd ed. (Upper Saddle River, NJ: Pearson Prentice Hall, 2006), 42.

3. The Blum-Blum-Shub Generator

The Blum-Blum-Shub[20] (BBS) secure pseudorandom bit generator (PRGB) is one of the most popular and widely used secure PRBGs. Named after Lenore Blum, Manuel Blum and Michael Shub, it is sometimes also referred to as the quadratic residue generator.[21] The base for the BBS is two large primes, *p* and *q*, that both are congruent to 3 modulo 4 and a randomly chosen number, *x*, which is relatively prime to the product *n* of the two primes. This random number is used to generate a seed to the generator by squaring it and taking the result modulo *n*.

$$p, q \equiv 3 \pmod 4, \quad (p \text{ and } q \text{ are large primes})$$

$$n = p \cdot q$$

$$x, \text{ realatively prime to } n$$

$$x_0 = x^2 \bmod n, \text{ generates as seed}$$

Pseudorandom bits are then generated by taking the square of the seed modulo *n* and using the least significant bit (which is the same thing as taking the result modulo 2). For the following bit the resulting number of the computation in the previous step is used instead of the seed. This process is repeated until required number of bits has been achieved.

$$x_i = \left(x_{i-1}^2 \bmod n\right) \bmod 2$$

The BBS generator is considered to be a generator secure for cryptographic purposes. Compared to other generators it is, however, slow since it uses complex calculations with large numbers. Its security is based on the Composite Quadratic Residues problem.[22]

---

[20] Leonore Blum, Manuel Blum and Michael Shub, "A Simple Unpredictable Pseudo-Random Number Generator," *SIAM Journal on Computing*, 15 (May. 1986) 364–383.

[21] Trappe and Washington, *Introduction to Cryptography with Coding Theory*, 42.

[22] Stinson, *Cryptography Theory and Practice*, 338.

4. Other Generators

There are a number of other pseudorandom number/bit generators available. Two examples of generators considered to be secure for cryptographic purposes are the RSA pseudorandom bit generator[23] and the Micali-Schnorr pseudorandom bit generator.[24] The ANSI X9.17 pseudorandom bit generator[25] and the FIPS 186 pseudorandom number generator for DSA private keys[26] are considered insecure for cryptographic purposes but are still useful for many other areas, such as simulation, gaming, etc.

C. IMPROVING RANDOMNESS IN SEQUENCES

As previously mentioned, not all ways of generating random numbers are good enough. There are, however, ideas for improving the randomness of the output from a pseudorandom number generator using different types of techniques. Two such techniques are the shrinking and the self-shrinking generator. These are normally used in combination with linear feedback shift registers, but we will later apply the self-shrinking generator on some strings of other, pseudorandomly, generated bit strings.

1. The Shrinking Generator

The concept of the shrinking generator was first published in 1993 by Coppersmith, Krawczyk and Mansour.[27] The main idea is to run two LSFRs (*R1* and *R2*) in parallel using the same clock so that their outputs are generated at the same time. The two LSFRs are however using different irreducible polynomials[28] in their construction and therefore generate outputs independent from each other. If at any clock cycle the output bit from LSFR *R1* is a "1," the corresponding output bit from LSFR *R2* is used as

[23] Alfred J. Menezes, Paul C. Van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography* (Boca Raton, FL: CRC Press, 1997), 185–186.

[24] Ibid.

[25]Menezes et al., *Handbook of Applied Cryptography*, 173–175.

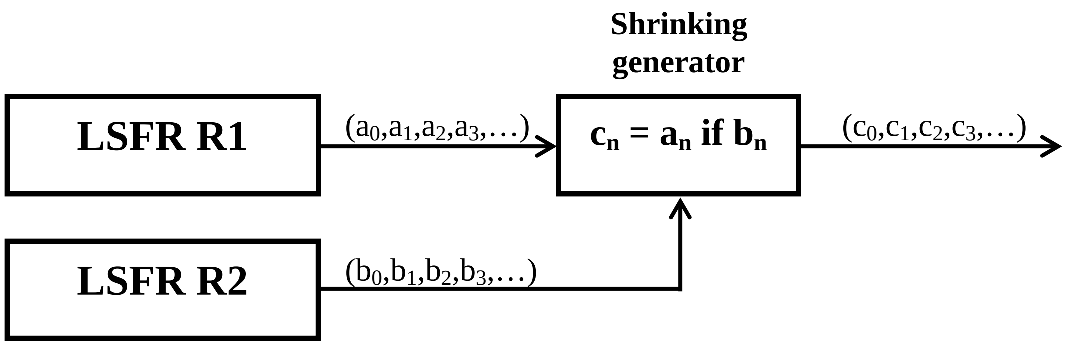
[26] Ibid.

[27] Don Coppersmith, Hugo Krawczyk and Yishay Mansour, "The Shrinking Generator," *Advances in Cryptology - CRYPTO '93* (Lecture Notes in Computer Science (LNCS), Vol. 773), Santa Barbara, CA: Springer 1993), 22–39.

[28] See Appendix B.

an output of the shrinking generator. If the output bit from LSFR *R1* instead is a “0,” the corresponding output bit from LSFR *R2* is discarded and not used as an output of the shrinking generator.[29] A simpler, but maybe not so random way, to achieve this is to create a random string of “1’s” and “0’s” and use this as a template. When this template is used as an overlay to an output of a LSFR (or any string) every bit in the generated string corresponding to the position of a “1” in the template is used as an output from the shrinking generator while bits corresponding to the positions of the “0’s” are discarded.

The result of the shrinking generator will be a string shrunken to about half its original length since half of the output bits from the LSFR R1 are “1’s” and the other half are “0’s” (or actually one more “1” than “0’s”[30]). The properties of the resulting output from the shrinking generator might however be different since any patterns or statistical properties in the output of LSFR *R2* now have been changed.



```
LSFR R1:   1101000110010111011001101100110111010111
LSFR R2:   0110100101101001011010010110100101101001
Output:     10 0  1 00 0  1 11 0  0 10 1  1 10 0  1
```

Figure 5. The shrinking generator and an example of an output.

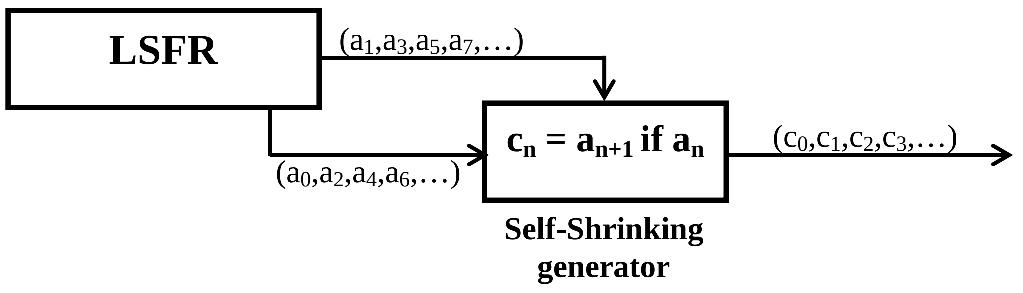
[29] Menezes et al., *Handbook of Applied Cryptography,* 211–212.

[30] See Linear Feedback Shift Registers in Chapter III, Section B. 1.

2. The Self-shrinking Generator

Closely related to the shrinking generator is the self-shrinking generator. The latter differs from the first in the way that it uses only one LSFR instead of two. However, it still results in an output depending on the positions of the “1’s” in the string.

In the self-shrinking generator the input string is being partitioned into pairs of bits. These pairs will then be one of the following combinations: “00,” “01,” “10” or “11.” If the first bit in these pairs is a “1” the second bit will be used as an output of the generator. On the other hand, if the first bit is a “0” the second bit will be discarded.[31] All the first bits are used in the decision-making only and will be discarded. If an LSFR is used as an input to the self-shrinking generator the likeliness of the four bit combinations is equal since “1’s” and “0’s” appear at the same rate in the output of the LSFR. The output of the self/shrinking generator might however have different statistical properties.


LSFR
$(a_1,a_3,a_5,a_7,\ldots)$
$(a_0,a_2,a_4,a_6,\ldots)$
$c_n = a_{n+1}$ if a_n
$(c_0,c_1,c_2,c_3,\ldots)$
Self-Shrinking generator

```
LSFR:   11 01 00 01 10 01 01 11 01 10 01 10 11 00 11 01 11 01
Output:  1           0        1     0     0  1     1     1
```

Figure 6. The self-shrinking generator and an example of an output.

[31] Menezes, et al., *Handbook of Applied Cryptography*, 221.

D. ENTROPY

As mentioned above, all pseudorandom number generators need an input, or a seed, to start generating numbers. If this input is not chosen at random the whole sequence generated is also not random, since any pseudorandom number generator is deterministic. If the input is known (or can be guessed) the output will also be known (i.e., the generated bit string lacks random properties). Therefore, it is important to use truly random numbers as input. This is done through the use of entropy.

Entropy can be defined as a level of uncertainty of predicting a value or as NIST states: "Entropy is defined relative to one's knowledge of *X* prior to an observation and reflects the uncertainty associated with predicting its value—the larger the entropy, the greater the uncertainty in predicting the value of an observation."[32] We therefore need a source that can take a number of different states; these states can then be discretized and used as a random input. If the number of possible states is low we will receive very low entropy. The same goes if the likeliness of the source taking a certain state differs a lot from the other states; then the entropy will also be low. One can compare it to a raffle with numbered tickets in it. If there are only a five tickets in it one is more likely to predict which ticket will be drawn than if there are 500 tickets (i.e., the entropy is higher with more tickets). In a similar way; if there were 500 tickets in a raffle but 250 of them had the same number one would be more likely to be able to predict the winning number (i.e., the entropy decreases if the probabilities for the outcomes are not equal). The recommendations on an entropy source according to NIST are as follows:[33]

To create an entropy source we need first and foremost a noise source. The reason for utilizing noise is that it is often the only truly non-deterministic source we have available. The noise can be achieved from a number of different sources such as capacitor discharging time, time differences between key strokes and mouse movements. In mobile devices sources like the camera lens, the accelerometer and radio signal strength could be

[32] Elaine Barker and John Kelsey, *Recommendation for the Entropy Sources Used for Random Bit Generation* (National Institute of Standards and Technology (NIST) DRAFT Special Publication 800–90B), (Gaithersburg, MD: U.S. Department of Commerce, 2012), 19.

[33] Ibid.

used. The noise source then needs to be digitalized to be of any use in a computational algorithm, but this is often easily done.

After the noise source has been digitalized one can choose to apply a conditioning component. This component helps to avoid the output being biased and increases the entropy rate. The use of this conditioning component is not required but might be needed depending on the noise source and its characteristics. For further discussions regarding the conditioning component, please read the NIST special publication.[34]

The last important part of an entropy source is health testing. We trust the entropy source to give us random output, but we still have to check that it is working the way it is supposed to. Therefore, a health test on the entropy source must be performed. These tests can be performed as startup tests, continuous tests and on-demand tests and should not only check the noise source itself but also the digitalization and the conditioning component (if applicable). Since we put so much trust in the entropy source we have to be able to detect any deviations, biases or malfunctions as soon as possible and with high probability. Health tests can also help us identify common failure modes and make it possible to correct for these using, for example, a conditioning component.

So, if we now have an entropy source that supplies us with as pure random numbers as possible, why do we bother using pseudorandom number generators? Why not use the output of the entropy source? The answer is quite simply time. While an entropy source requires quite some time to collect a usable amount of data a pseudorandom number generator can generate the same amount much faster. As previously mentioned, however, it requires a truly random input, or seed, to generate a bit string that has properties that are random enough.

[34] Barker and Kelsey, *Recommendation for the Entropy Sources Used for Random Bit Generation.*

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ATTACKS

As soon as a new encryption method is presented it is seen as a challenge to find feasible attacks on it. The same goes for bit strings generated by pseudorandom number generators. With an attack we mean to find a way to predict the output of the generator. It is also important to understand that the definition of an attack being successful is not necessarily that it is easy to predict an outcome but that the outcome can be predicted with higher probability than someone would need just by guessing. Strictly speaking, a lottery with 100 tickets ranging from the numbers 1 through 100 can be considered successfully attacked if we can show that no three-digit number can be drawn as the winning number, even if there then are 99 possible winning numbers left. Some possible attacks on pseudorandom number generators are presented here.

A. ATTACKS ON SHRINKING AND SELF-SHRINKING GENERATORS

A number of attacks have been developed on both the shrinking and the self-shrinking generator. Two of them are presented in the following sections. Others have been developed through the work of Kitae Jong[35] et al., Simon R. Blackburn[36] and Bin Zhang and Dengguo Feng[37] just to mention some.

1. Attack on Short Sequence Linear Feedback Registers Using the Self-Shrinking Generator

This attack uses the knowledge that the original (unknown) bit string is grouped into pairs of two bits; this is then compared to the output bit string (known). As mentioned in the discussion regarding the self-shrinking generator[38] we have four

[35] Kitae Jeong et al., "Improved Fast Correlation Attack on the Shrinking and Self-shrinking Generators," *Progress in Cryptology - VIETCRYPT 2006* (Lecture Notes in Computer Science (LNCS), Vol. 4341), (Hanoi, Vietnam: Springer, 2006), 260–270.

[36] Simon R. Blackburn, "The Linear Complexity of the Self-Shrinking Generator," *IEEE Trans. Inf. Theory*, 45 (September 1999), 2073–2077.

[37] Bin Zang and Denggou Feng, "New Guess-and-Determine Attack on the Self-Shrinking Generator," *Advances in Cryptology - ASIACRYPT 2006* (Lecture Notes in Computer Science (LNCS), Vol. 4284), (Shanghai, China, Springer: 2006), 54–68.

[38] For more information regarding the self-shrinking generator, see Chapter III, Section C. 2.

alternatives for each pair: 00, 01, 10, 11, each approximately just as possibly likely. When we know the first output bit, say 1, we also know that the first two bits for sure are not 10 (which would have given 0 as the first output bit). Furthermore we know that the probability for the very first bit to be 0 is equal to the probability for it to be 1. This combined leads us to the conclusion for the following probabilities for the first two bits: p(00) = ¼ , p(01) = ¼, p(10) = 0, p(11) = ½. This is then repeated for all pairs of bits and we can thereby assume an original bit string with higher probability than by just guessing.[39]

2. The Backtracking Algorithm

Another attack on LSFRs is the Backtracking Algorithm.[40] This algorithm requires that the feedback polynomial of the LSFR is known. It is based on an attack on the shrinking generator where the inner state of LSFR *R2* is guessed and used to create the *R2* sequence. Through this single bits of the *R1*-sequence can be reconstructed which all gives a linear equation. When enough bits have been recreated we can solve the linear equations and find the inner state of *R1*. This can then be double checked by running the two LSFRs and checking the output using the shrinking generator. A similar method can be applied to the self-shrinking generator. Now, however, since all even bits serve as the equivalent of the *R2*-sequence in the case with the shrinking generator and they are not the complete output of an LSFR, they are not necessarily linearly dependent (i.e., they have to be guessed bit by bit). This makes the attack on the self-shrinking generator more complicated and not so straightforward.

B. OTHER ATTACKS

There are also attacks that focus not on a specific algorithm or method but instead work directly with the string of random bits. One such powerful, and quite fascinating

[39] Erik Zenner, Matthias Krause, Stefan Lucks, “Improved Cryptanalysis of the Self-Shrinking Generator,” *Australasian Conference on Information Security and Privacy (ACISP) 2001* (Lecture Notes in Computer Science (LNCS), Vol. 2119), (Sydney, Australia: Springer, 2001), 21–35.

[40] Ibid.

algorithm, is the Berlekamp-Massey algorithm that finds a linear dependence in a string of apparently random bits.

1. The Berlekamp-Massey Algorithm

A string of “0’s” and “1’s” placed “randomly” may seem very random indeed. However, there is always a function that will express a linear dependency between the bits in any given string, even the truly random ones. In “poor” random strings such a function can easily be created to predict/compute the next bit. In a really poor random sting this equation is very simple; in not so poor random strings it is a bit more complicated. Such a function can also be created for the really good random strings. However, in the cases of the really good random strings, the equation needs the input of all the previous bits in the strings to predict/compute the very last bit.

The algorithm to create these “predicting functions” is called the “Berlekamp-Massey algorithm.”[41] Given a binary output sequence, the Berlekamp-Massey algorithm is used to find the simplest linear feedback shift register that creates this very same sequence. This algorithm walks through the binary string bit by bit, adding complexity to the function when needed to create the target bit string. An example of how it can be applied is presented as follows:[42]

Assume the bit string z_n of length 20 is observed; z_n = 11010110010001111010. It is very difficult to intuitively say whether or not the bits in this string have a linear dependency, but we will see that they are indeed linearly dependent and in a not too complex way. For the computations in the algorithm we need to keep track of a number of variables:

N	The current index, or the number of bits “taken into operation”
L_N	The complexity at a given index N
m	The largest index such that $L_m < L_N$

[41] Menezes et al., *Handbook of Applied Cryptography,* 200–202.

[42] An online calculator of the Berlekamp-Massey algorithm is available at http://bma.bozhu.me/

$f_N(x)$ The current function used

For every step in the algorithm the function $f_N(x)$ is kept unchanged unless it no longer gives a correct result for the last bit. When $f_N(x)$ must be recomputed the complexity L_N also must be recomputed and used as an input. The new complexity and function are computed using the following formulas for L_{N+1} and $f_{N+1}(x)$:

$$L_{N+1} = \max\left(L_N, (N+1) - L_N\right)$$

$$f_{N+1}(x) = x^{L_{N+1}-L_N} \cdot f_N(x) + x^{L_{N+1}-N+m-L_m} \cdot f_m(x)$$

To initiate the Berlekamp-Massey algorithm the first step is to at index $N=0$ setting the complexity to 0 ($L_0 = 0$) and the function to 1 ($f_0(x) = 1$). Thereafter at index $N=1$ computing the complexity and setting the function to be $x+1$, ($f_1(x) = x+1$):

$N = 0 \qquad z = \varnothing \qquad L_0 = 0$

$f_0(x) = 1$

$N = 1 \qquad z = 1 \qquad L_0 = \max\left(L_0, 1 - L_0\right) = \max(0,1) = 1$

$f_1(x) = x+1$

We will now try to find a function $f(x)$ that generates all the following bits. In this function we want terms to solve the binary equation $f(x) = 0$ where the powers of x indicates the indexes of the last bits in the string, e.g., in the string 110010, $x_4 = 0$ $x_3 = 1$ $x_2 = 0$ $x_1 = 0$ $x_0 = 1$. (Note that index 0 represents the term x^0, in the functions that follows this is simplified to a 1, since $x^0 = 1$). The first bits are not of interest from an indexation perspective.

In the next step the existing function $f_1(x)$ still works since $z = 11$ and $f_1(x) = 1 + 1 = 0$. The function and the complexity remain unchanged; $f_2(x) = f_1(x)$ and $L_2 = L_1$:

$$N = 2 \qquad z = 11 \qquad L_2 = 1$$

$$f_2(x) = x + 1$$

In the third step the function $f_2(x)$ is no longer applicable since $z = 110$ but $f_2(x) = 0 + 1 = 1 \neq 0$. We recompute the complexity L_3 and compute a new function $f_3(x)$ by applying the formula as described above:

$$N = 3 \qquad z = 110 \qquad L_3 = \max(L_2, 3 - L_2) = \max(1, 2) = 2$$

$$f_3(x) = x^{2-1} \cdot f_2(x) + x^{2-2+0-0} \cdot f_1(x) = x^1 \cdot (x + 1) + x^0 \cdot 1 = x^2 + x + 1$$

Double checking the function $f_3(x)$ above on z gives the result: $x^2 + x + 1 = 0 + 1 + 1 = 0$ so it is OK.

In the fourth step, the previous function works since $z = 1101$ and $f_3(x) = 1 + 0 + 1 = 0$. The index N is increased while the complexity, and the function remains unchanged; $L_4 = L_3$ and $f_4(x) = f_3(x)$:

$$N = 4 \qquad z = 1101 \qquad L_4 = 2$$

$$f_4(x) = x^2 + x + 1$$

In the fifth step we have $z = 11010$ but $f_4(x) = 0+1+0 \neq 0$ so we have to re-compute the complexity L_5 and the function $f_5(x)$:

$$N = 5 \qquad z = 11010 \qquad L_5 = \max(L_4, 5 - L_4) = \max(2,3) = 3$$

$$\begin{aligned} f_5(x) &= x^{3-2} \cdot f_4(x) + x^{3-4+2-1} \cdot f_2(x) = x^1 \cdot (x^2 + x - 1) + x^0 \cdot (x+1) = \\ &= (x^3 + x^2 + x) + (x+1) = x^3 + x^2 + 2x + 1 = x^3 + x^2 + 1 \end{aligned}$$

In the sixth step we have $z = 110101$ and $f_5(x) = 1+0+0 = 1 \neq 0$, Again, the index *N* is increased and both complexity L_6 and the function $f_6(x)$ are recomputed:

$$N = 6 \qquad z = 110101 \qquad L_6 = \max(L_5, 6 - L_5) = \max(3,3) = 3$$

$$\begin{aligned} f_6(x) &= x^{3-3} \cdot f_5(x) + x^{3-5+4-2} \cdot f_4(x) = x^0 \cdot (x^3 + x^2 - 1) + x^0 \cdot (x^2 + x + 1) = \\ &= (x^3 + x^2 + 1) + (x^2 + x + 1) = x^3 + 2x^2 + x + 2 = x^3 + x \end{aligned}$$

In the seventh step, $z = 1101011$ but $f_6(x) = 1+0 = 1 \neq 0$. Once again we increase index *N* and recompute the complexity L_7 and the function $f_7(x)$:

$$N = 7 \qquad z = 1101011 \qquad L_7 = \max(L_6, 7 - L_6) = \max(3,4) = 4$$

$$\begin{aligned} f_7(x) &= x^{4-3} \cdot f_6(x) + x^{4-6+4-2} \cdot f_4(x) = x^1 \cdot (x^3 + x) + x^0 \cdot (x^2 + x + 1) = \\ &= (x^4 + x^2) + (x^2 + x + 1) = x^4 + 2x^2 + x + 1 = x^4 + x + 1 \end{aligned}$$

In the eighth step, $z = 11010110$ while $f_7(x) = 0+0+1 = 1 \neq 0$. Yet again we increase the index N and recompute the complexity L_8 and the function $f_8(x)$:

$$N = 8 \qquad z = 11010110 \qquad L_8 = \max(L_7, 8 - L_7) = \max(4,4) = 4$$

$$f_8(x) = x^{4-4} \cdot f_7(x) + x^{4-7+6-3} \cdot f_6(x) = x^0 \cdot (x^4 + x + 1) + x^0 \cdot (x^3 + x) =$$
$$= (x^4 + x + 1) + (x^3 + x) = x^4 + x^3 + 2x + 1 = x^4 + x^3 + 1$$

In the ninth step, $z = 110101100$ and $f_8(x) = 1+0+1 = 0$, so the function works and $f_9(x) = f_8(x)$.

In the tenth step, $z = 1101011001$ and $f_9(x) = 1+0+1 = 0$, so the function works and $f_{10}(x) = f_9(x)$.

As a matter of fact, we have now found the function that works for the complete observed bit string. Therefore, no further computation is necessary for either the complexity L_N or the function $f_N(x)$. Remaining bit values can be controlled with the function achieved:

N	z	$f(x) = x^4 + x^3 + 1$
9	110101100	0 + 0 + 0 = 0
10	1101011001	1 + 0 + 1 = 0
11	11010110010	0 + 1 + 1 = 0
12	110101100100	0 + 0 + 0 = 0
13	1101011001000	0 + 0 + 0 = 0
14	11010110010001	1 + 0 + 1 = 0
15	110101100100011	1 + 1 + 0 = 0
16	1101011001000111	1 + 1 + 0 = 0
17	11010110010001111	1 + 1 + 0 = 0
18	110101100100011110	0 + 1 + 1 = 0
19	1101011001000111101	1 + 0 + 1 = 0
20	11010110010001111010	0 + 1 + 1 = 0

Table 2. Checking applicability of the function $f_N(x)$ for steps $N = (9 \ldots 20)$.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TEST RESULTS

For the tests a number of bit strings have been used. Time constraints and difficulties getting access to usable data have made it challenging to get all tests done in time, and a more thorough examination could be done in the future. The tests performed, however, show some interesting results.

A. PERFORMANCE OF THE TESTS

The string of pseudorandom bits was run through the NIST test suite. Since the test suite requires multiple strings to get usable statistics the strings used were divided by the NIST test program into a number of equally long bit streams. Hereafter the term “bit string” will be used for the original pseudorandom generated string used as the general input for the NIST test suite, while the term “bit stream” will be used for the input to the NIST subtests. The bit streams created were then run through all the tests in the test suite, and the result was saved. The original bit string was then modified by running it through a self-shrinking generator.[43] The resulting string was then run through the same NIST tests again. It is important to notice that this new bit string was now shorter than the original (the SSG is expected to shorten a truly random string to one fourth of its original length). To get comparable results the new bit string was divided into the same number of bit streams that were used in the original test. The new bit streams were therefore shorter than the originals, but they reflected the information retrieved from the full length bit streams in the original bit string since they were created using them as input. The results after having run the NIST test suite on the string after having applied the self-shrinking generator was compared to how the original, non-modified, bit string performed on the NIST test suite. Depending on which original source was being used, the NIST bit stream length and the number of bit streams tested varied.

[43] See python script for the self-shrinking generator in Appendix D.

B. TESTING A STRING GENERATED BY AN ANDROID PHONE

Since not as much work has been put into pseudorandom number generators for mobile devices as it has for their stationary equivalents, it was expected that some non-random properties would be discovered. The intention was then to improve the randomness of the generated bit string by applying the self-shrinking generator (SSG). Generating bit strings on an Android phone was outside the scope of this thesis, and access to such strings turned out to be more limited than expected. Further data could therefore render more information. It was not until very late in the research that the bit strings needed were made available, and the time available for tests was very limited. Some tests should be run again with other parameters to see if the results could be affected.

For these tests (which are described in Chapter II), two strings of random bits generated on an Android phone were used.[44] One was created using the /dev/urandom block device (hereafter referred to as string A) while the other one came from the SecureRandom function provided by Java (hereafter referred to as string B). They were presented in a pure binary file and were therefore converted[45] to an ASCII representation so that they could be modified using the SSG script. The conversion script also cut the file to a manageable length since the NIST test suite would not accept a too great input.[46] For the tests, 100 bit streams of length 10^6 were used from each string.

1. Results before Applying the SSG

NIST recommendations require a 96% pass rate for the bit streams, and this was fulfilled for string A, even if not all bit streams passed all subtests. String B passed all tests with one exception. In Test #7, "The Non-overlapping Template Matching Test," for one of the 148 templates tested only 95 of 100 bit streams passed resulting in a 95% pass rate. However, this must be considered such a rare event that no conclusions of the

[44] The strings were made available through another research project.

[45] See python script for binary to ASCII conversion in Appendix C.

[46] The maximum input used in the test was approx. 750MB. Suspected file size limit is 1GB.

original bit string being non-random can be drawn. Both strings must therefore be considered as fulfilling the NIST recommendations for required randomness. Since non-random properties were expected, this was a bit of a disappointment. The SSG was applied anyway to see if the pass rate could be improved further.

2. Results after Applying the SSG

After the SSG had been applied, the bit strings A and B were run through the NIST test suite again. This time, both bit strings passed all the tests. The SSG can then at least be considered to have improved the result regarding one matching template in string B. It is however doubtful whether there was an overall improvement. The pass rate did improve for some tests, but it remained constant or even deteriorated for others. The changes in pass rate were so low that no conclusions could be drawn from this test.

C. TESTING A STRING GENERATED BY LINUX

As a reference a pseudorandom bit string generated by the /dev/urandom function in Linux (hereafter referred to as string C). The /dev/urandom function was chosen in an attempt to generate a string that lacked some random properties. The /dev/random function could have been used, but it was expected to perform better in the tests. String C was also converted to ASCII representation before being run through the NIST test suite and the SSG. In this test we used 300 bit streams of length 10^6. The original string was not as long as the Android strings, and after having applied the SSG, the bit stream length had to be shortened to $2.5 \cdot 10^5$ to still be able to test 300 bit streams. The original string C performed equally well as the Android bit strings and passed all tests with an equivalent pass rate.

A divergent result was achieved when the bit string was tested after the SSG had been applied. The string now passed all tests with an acceptable pass rate. However, it failed to pass Test #9 from the perspective of an even distribution of P-values. It seems like the test requirements for input variables were not met. This problem has not been detected with any other string. Why the requirements were not met with this string in this test is not clear.

D. TESTING A STRING WITH POOR RANDOM PROPERTIES

As a reference, a string of “really poor” bits was tested. Even if it is easy to find or create a string with really poor random properties (e.g., all “1’s” or a repetitive pattern of a given length) it remains difficult to find a string of bits with some random properties but still not random enough to be anywhere near to pass the NIST test suite. To create such a string, we used the U.S. Constitution with its 27 amendments.[47]

1. Creating a Bit String with Poor Random Properties

The string to be tested was created by letting the characters and spaces of the Constitution be represented by their 8-bit ASCII representation while all line breaks were removed. The result was an approximately 350 000 bit long string. This string now had some random properties while it lacked others. It, for example, passed the linear complexity test. This is not surprising since the Berlekamp-Massey algorithm[48] has shown us how the full length of a string has to be taken into account to compute the linear complexity. Since the characters constantly change in a text, this change affects the complexity. The string created, however, did not pass the most basic test, the frequency test. This is what could be expected. Letters A-Z are represented by 01000001–01011010 and a-z by 01100001–01111010, (i.e., bit #3 is always is a “0” for majuscule (capital) letters and a “1” for all minuscule letters). Since there are many more minuscule than majuscule letters in a regular text this will result in a higher frequency of “1’s” than “0’s” in the created string. Furthermore, in a regular text the letter “o” (ASCII: 01101111, 6 “1’s” 2 “0’s”) occurs more often than the letter “b” (ASCII: 01100010, 3 “1’s” 5 “0’s”) resulting in a bias towards a higher frequency of “0’s.”[49] These are just two examples of what affects bit frequency in a string derived from a text. Since all characters are represented by eight bits and they all start with a zero, the NIST tests will notice this as in, for example, “The Binary Matrix Rank Test” (Test #6) where smaller matrices created

[47] Available at: www.usConstitution.net/const.txt.

[48] See Chapter IV, Section B.1.

[49] For more information on the distribution of letters in a text search for information on “letter frequency.”

by the string are tested for the linear dependence. Furthermore, in a string of this size you would expect to find some long runs of “1’s” and “0’s.” But since the character represented by all “0’s” is the “NUL” and the character represented with all “1’s” is “ÿ” (both being extremely rare in most texts, especially in the U.S. Constitution) there will certainly be no run longer than 14 (2·[8–1]) bits.

2. Test Results

For testing the string described in the previous paragraph, it was split into 10 bit streams, each of length 35 000 bits. In another test 100 bit streams of length 3 500 bits were also tested, but these bit streams seemed too short to result in any interesting results. When being run through the NIST test suite, the bit string created, as expected, did not pass the NIST test for randomness.[50] Even after the self-shrinking generator was applied the bit string did not pass the test suite. It is r interesting, though, to see what improvements were made and why.

Since the string did not pass the frequency test we could not expect it to pass after the SSG had been applied. That is because the relative frequency generally is maintained by the SSG. This also affects the result in the tests directly based on the frequency of bits (Tests #2 through #5) and makes the string fail those tests as well. However, by applying the SSG we seem to have improved the result for Test #6, “The Binary Matrix Test.” By applying the SSG we have to a great extent eliminated the property of the original string where every character was represented by eight bits. Since characters A-Z in ASCII representation all start with 0100 or 0101, all bits from the first half of the ASCII representation are being discarded. For the characters a-z (starting with 0110 or 0111) however, the second pair of bits results in on bit (1 or 0) in the resulting string. Thus, a minuscule character will always contribute to one bit more than its majuscule equivalent. From the second half of the ASCII representation (bits #5 through #8) varying numbers of bits are maintained. After having applied the SSG no pattern revealing the original 8-

[50] We can now say, therefore, that we have proven that the U.S. Constitution was not written at random (if anyone ever might have suspected it to have been).

bit “partition” can be seen. The characters have instead resulted in 0–3 bits in the new bit string.

By breaking up the 8-bit “partition” we have also made it more difficult for “Template Matching Tests” to find non-randomness (i.e., it is more difficult to find these matching templates once we have substituted every 8-bit character representation with 0–3 bits). All the bit streams of the new string did not pass all matching templates tests, and the string as such did not pass the test in full. Even so, we can observe a 500% improvement for the string in passing the subtests after the SSG was applied as compared to before.

The original string did pass Test #10, “The Linear Complexity Test.” An improvement of the distribution of the p-values can also be seen. This can be seen as a result of the SSG “removing” the 8-bit partition of the original string. A string that has less obvious partitioning will result in more varied complexity.

VI. CONLUSIONS

It was expected that pseudorandom bits strings generated on an Android phone would lack some random properties. An attempt would then be made to improve these flaws in randomness by applying the self-shrinking generator (SSG).

A. TESTED STRINGS

Tests using the NIST test suite (described in Chapter II) showed that the bit strings generated on the Android phone passed the NIST tests with a pass rate according to NIST recommendations. Applying the SSG affects the test results, but no conclusions can be made whether it is for better or for worse.

The test performance of the Android-generated strings was compared to the performance of a Linux-generated string. They performed equally well, and no obvious differences could be identified.

Tests were also run on a string with poor random properties, a plain text in ASCII representation. This string drastically failed all tests except the linear complexity test. After having applied the SSG, this string showed improved results in four of the tests: the "Binary Matrix Rank Test," "Discrete Fourier Transform (Spectral) Test," "Non-overlapping Matching Template Test" and "Overlapping Template Test." It also still passed the "Linear Complexity Test."

B. TEST ALGORITHM AND THE SELF-SHRINKING GENERATOR

After having run the NIST test suite, before and after the SSG had been applied, on a bit string with poor random properties, such as the plain text string, it was clear that there are two main aspects of random properties that are being tested by the NIST test suite: frequency and linear dependency.

In a string with good random properties the frequency of 1's and 0's should be about equal. This is not necessarily true in a string with poor random properties, and definitely not so in a string based on a binary ASCII representation of an English text.

The character frequency in English combined with the binary ASCII representation of the characters results in a string having the ratio 45/55 of 1's and 0's. Since the SSG compresses the original string with guidance on the occurrence of 1's as the first bits in a pair of bits, the probability for the second bit in these pairs being a 1 or a 0 will be reflected in the new string created. Therefore, the ratio of the frequency of 1's and 0's in the resulting string is not 50/50, it has after the SSG was applied actually changed to a ratio of 38/62. This change in ratio occurs since the bit combination "10" (resulting in a single 0 in the SSG output string) is more common than the bit combination "11" (resulting in a single 1). The reason for this is that minuscules (lower case letters) are in majority in a text, and the majority, and the most frequently used, of the minuscules (a-o) are represented by "10" as bits 3 and 4, resulting in a single 0 in the SSG output string. The SSG is therefore for no much use when trying to improve test results from tests that are based on an analysis of the frequency of bits.

A string with good random properties is expected to have a great level of linear independency. A string created from a binary ASCII representation of a text will at every eighth bit start representing a new letter. This is a pattern, or a linear dependency, that will easily be detected in a test. The easiest way to understand this is to realize that the most frequently used characters used in a text are the letters A-z. The binary ASCII representation of these letters all start with the bits "01," i.e., two out of eight bits for every character can be predicted. This creates an obvious linear dependency in longer bit strings. Applying the SSG removes the two initial bits in the binary ASCII representation of all letters. It also affects the following bits resulting in an output of 0–3 bits for every 8-bit representation of a letter. After the SSG has been applied to a string, its test results in the NIST test suite dramatically improves for tests based on linear dependence and spectral tests, except for the linear complexity test where already the original string performs well. The reason for this being that in this test the full length of the string is being analyzed instead of for example the "Binary Rank Matrix Test" where only blocks of the string are being tested. Applying the SSG on a string can therefore improve randomness from a perspective of linear dependency in a string.

C. SUGGESTIONS FOR FURTHER RESEARCH

Time constraints made it difficult to perform as many tests under varied input condition as would have been desirable. A more thorough testing and analysis of test results, before and after having applied the self-shrinking generator could therefore be recommended to confirm the results presented here. More strings with limited random properties of various degrees could also be examined to better identify the self-shrinking generator's capability of improving randomness in a string. Furthermore, more research could be focused on the mathematics behind how the self-shrinking generator affects a string. This thesis inly notes that it affects the randomness of the string, not so much why.

When it comes to the generator used, this thesis only examines the self-shrinking generator. The shrinking generator is presented but due to time constraints not examined. A drawback with the self-shrinking generator is that it shortens the input string to approx. 25% of its original length while the shrinking generator shortens it to only approx. 50%. The drawback with the shrinking generator is that it needs two inputs, a string to be shrunken, and a string to use as a template for performing this shrinking. There could however be an improved self-shrinking generator, which for example does not pair the bits in the input string as the existing self-shrinking generator does but instead uses every bit as a decision bit to decide whether the following bit should be discarded or be a bit in the resulting output string. A decision rule that uses multiple bits in the original string to decide the output could also be applied. This would be like using the string itself as a template in a shrinking generator and give a greater output string (like the shrinking generator) without a template sting (like the self-shrinking generator). These types of expansion of the shrinking generators have not been examined in this thesis but it could be interesting to do so.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATHEMATICAL FIELDS

In cryptology we normally use binary numbers and finite fields. A field is defined as a set S of elements under the operations addition and multiplication fulfilling the following properties:[51,52]

Closure	If $a,b \in S$, then $a+b$ and $a \cdot b \in S$
Associativity	If $a,b \in S$ then $a+\left(b+c\right)=\left(a+b\right)+c$ and $a\cdot\left(b\cdot c\right)=\left(a\cdot b\right)\cdot c$
Commutativity	$a+b=b+a$ and $a\cdot b=b\cdot a$
Identity	$a+0=a$ (additive identity)
	$a\cdot 1=a$ (multiplicative identity)
Inverses	$a+\left(-a\right)=0$ (additive inverse)
	$a\cdot a^{-1}=1$ (multiplicative inverse)
Distributivity	$a\cdot\left(b+c\right)=a\cdot b+a\cdot c$

If the number of elements in the field is *p* (a prime) they form a field under addition and multiplication modulo *p*. In cryptology we normally use fields of order two ($p=2$, binary), or in other words a Galois Field of degree two, $GF\left(p\right)=GF\left(2\right)$, or some of their extensions (see Appendix B).

[51] Neal H. McCoy and Gerald J. Janusz, *Introduction to Abstract Algebra* (Ann Arbor, MI: Trustworthy Communications, 2009), 3–5.

[52] Ibid. 71.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. FIELD EXTENSION

A field of order two does not give us many possibilities since it only has two elements. Therefore, we perform a field extension. By creating a polynomial of degree *n* using the elements of the field *S* as coefficients we achieve a field extension, $GF\left(p^n\right) = GF\left(2^n\right)$.

Example:

Using a polynomial of degree $n = 3$ we achieve an extension $GF\left(2^3\right)$ of the binary field $GF\left(2\right)$ which is represented by the polynomial $c_2x^2 + c_1x^1 + c_0x^0$ where the coefficients c_n are in the set $S = \left\{0,1\right\}$. This polynomial now gives us the possibility to express eight ($2^3 = 8$) different polynomials:

Polynomial	Polynomial full format	Coefficients, c_n
0	$0 \cdot x^2 + 0 \cdot x^1 + 0 \cdot x^0$	0 0 0
1	$0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$	0 0 1
x	$0 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$	0 1 0
$x+1$	$0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$	0 1 1
x^2	$1 \cdot x^2 + 0 \cdot x^1 + 0 \cdot x^0$	1 0 0
x^2+1	$1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0$	1 0 1
x^2+x	$1 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$	1 1 0
x^2+x+1	$1 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$	1 1 1

Table 3. Polynomials and their coefficients for the extended field $GF\left(2^3\right)$.

A polynomial, $q(x)$, is irreducible if there does not exist any other two polynomials, $p\left(x\right)$ and $g\left(x\right)$, such that $p\left(x\right) \cdot g\left(x\right) = q\left(x\right)$ (all polynomials being of degree greater than 0). The easiest way to check that a polynomial is irreducible is to check that the polynomials do not result in zero when $x = 0$ or 1:

$f(0) \neq 0$ and $f(1) \neq 0$

Remember that in binary $1+1=0$. Looking at the polynomials presented in the example above, x^2+x is not irreducible (since $1^2+1=0$), while x^2+x+1 is irreducible (since $1^2+1+1=1\neq 0$). The property of a polynomial being irreducible plays an important role in cryptology. In this example a polynomial of degree 3 has been used. Normally, polynomials of much greater degree are used.

APPENDIX C. PYTHON SCRIPT FOR CONVERTING A BINARY FILE TO BINARY ASCII REPRESENTATION

```
# This script reads and converts 100 bytes of binary data
# and converts it to a binary ASCII string of maximum
# length 800MB

import sys, binascii
f = open(sys.argv[1], “rb”)         # Open input file
g = open(“result,”“w”)              # Open output file

try:
  numbytes = 0                      # Set and index
  while True:
    bytes = f.read(100)             # Read 100 bytes
    numbytes = numbytes + 100       # Index increment
    if bytes == ‘‘ or numbytes > 10**8:
      # The line above limits output to 800MB
      break
    x = bin(int(‘1’+binascii.hexlify(bytes),16))[3:]
      # The line above converts to binary ASCII
      # The ‘1’+ prevents loosing leading 0’s
    g.write(x)                      # Write to output file

finally:
  f.close()                         # Close input file
  g.close()                         # Close output file
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. PYTHON SCRIPT FOR APPLYING THE SELF-SHRINKING GENERATOR TO A STRING

```
# Opens a file, applies the self-shrinking generator
# (if bit #1 is a “1” use bit #2 otherwise don’t use any of
# them, repeat for #3 and #4, etc.) and writes the result
# to a new file.

import sys, os

file = sys.argv[1]
f = open(file)           # Open input file
g = open(file+’mod,’”w”) # Create output file

statinfo = os.stat(file) # Identify string length
z = int(statinfo.st_size)

index = 1                # Limits number of loops
while (index  < z):      # to file length

  x=f.read(1)            # Read first bit
  index = index +1

  if x==“0”:             # If bit is 0 don’t
    y=f.read(1)          # use next bit
    index = index +1
  elif x==“1”:           # If bit is 0 use next bit
    y=f.read(1)
    index = index +1
    g.write(y)           # Write to output file
  else:                   # For unexpected inputs
    print “Unexpected input”
    f.close()
    g.close()
    quit()

f.close()                 # Close input file
g.close()                 # Close output file
quit()
```

LIST OF REFERENCES

Adamy, David L. 2004. *EW102, A second course in electronic warfare*. Boston, MA: Artech House Publishers.

Blackburn, Simon R. ‘The Linear Complexity of the Self-Shrinking Generator.” *IEEE Trans. Inf. Theory,* 45 (September 1999).

Blum, Leonore, Manuel Blum and Michael Shub. “A Simple Unpredictable Pseudo-Random Number Generator.” *SIAM Journal on Computing*, 15 (May 1986).

Breuer, William B. 1993. *Hoodwinking Hitler, the Normandy deception.* Westport, CT: Praeger Publishers.

Clausewitz von, Carl. 2002. *Om Kriget.* Stockholm, Sweden: Bonnier Fakta Bokförlag AB.

Coppersmith, Don, Hugo Krawczyk and Yishay Mansour. “The Shrinking Generator.” *Advances in Cryptology - CRYPTO ‘93.* Lecture Notes in Computer Science (LNCS), Vol. 773 (1993).

Jeong, Kitae et al. 2006. “Improved Fast Correlation Attack on the Shrinking and Self-shrinking Generators.” *Progress in Cryptology - VIETCRYPT 2006 Lecture Notes in Computer Science (LNCS),* Vol. 4341.

Knuth, Donald E. 1981. *The art of computer programming, volume 2/Seminumerical programming*. Reading, MA: Addison-Wesley.

Leon-Garcia, Alberto. 1994. *Probability and random processes for electrical engineering.* Reading, MA: Addison-Wesley.

McCoy, Neal H. and Gerald J. Janusz. 2009. *Introduction to abstract algebra.* Ann Arbor, MI: Trustworthy Communications.

Menezes, Alfred J., Paul C. Van Oorschot and Scott A. Vanstone. 1997. *Handbook of applied cryptography*. Boca Raton, FL: CRC Press.

Singh, Simon. *The code book.* 1999. New York, NY: Random House.

Stallings, William and Lawrie Brown. 2008. *Computer security, principles and practice.* Upper Saddle River, NJ: Pearson Educational.

Stinson, Douglas R. 2006. *Cryptography theory and practice.* Boca Raton, FL: Chapman & Hall/CRC.

Trappe, Wade and Lawrence C. Washington. 2006. *Introduction to cryptography with coding theory*. Upper Saddle River, NJ: Pearson Prentice Hall.

Tzu, Sun. 1991. *The art of war.* Boston, MA: Shambhala Publications.

U. S. Department of Commerce. 2012. *Recommendation for the Entropy Sources Used for Random Bit Generation.* By Elaine Barker and John Kelsey. National Institute of Standards and Technology (NIST) DRAFT Special Publication 800–90B.

———. 2010. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Application*, By Andrew Ruhkin et al. National Institute of Standards and Technology (NIST) Special Publication 800–22 Rev. 1a.

U. S. Joint Chiefs of Staff. 2011. Joint Publication 3-13.2. *Military Information Support Operations*.

Waltz, Edward. 1998. *Information warfare, principles and operations.* Norwood, MA: Artech House, Inc.

Zang, Bin and Denggou Feng. “New Guess-and-Determine Attack on the Self-Shrinking Generator,” *Advances in Cryptology – ASIACRYPT 2006 Lecture Notes in Computer Science (LNCS),* Vol. 4284, (2006).

Zenner, Erik, Matthias Krause, Stefan Lucks. 2001. “Improved Cryptanalysis of the Self-Shrinking Generator.” *Australasian Conference on Information Security and Privacy (ACISP) 2001 Lecture Notes in Computer Science (LNCS),* Vol. 2119.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California